Ptolemy Ptutorial

Edward A. Lee

UC Berkeley

This tutorial is aimed at the software developer planning to use Ptolemy as a platform.

For Ptolemy version 0.7.

Copyright © 9/24/97,
The Regents of the University of California
All rights reserved.

Ptolemy as a Tool and as a Laboratory

Ptolemy is

- **Extensible**
- **Publicly available**
- **An open architecture**
- **Object-oriented**

Allows for experiments with:

- **Models of computation**
- **Domain-specific tools**
- **Design methodology**
- **Software synthesis**
- **Hardware synthesis**
- **Cosimulation**
- **Codesign**

Rationale for heterogeneity: specialized models are

- **More useful to the system-level designer**
- **More amenable to hardware and software synthesis.**

Documentation

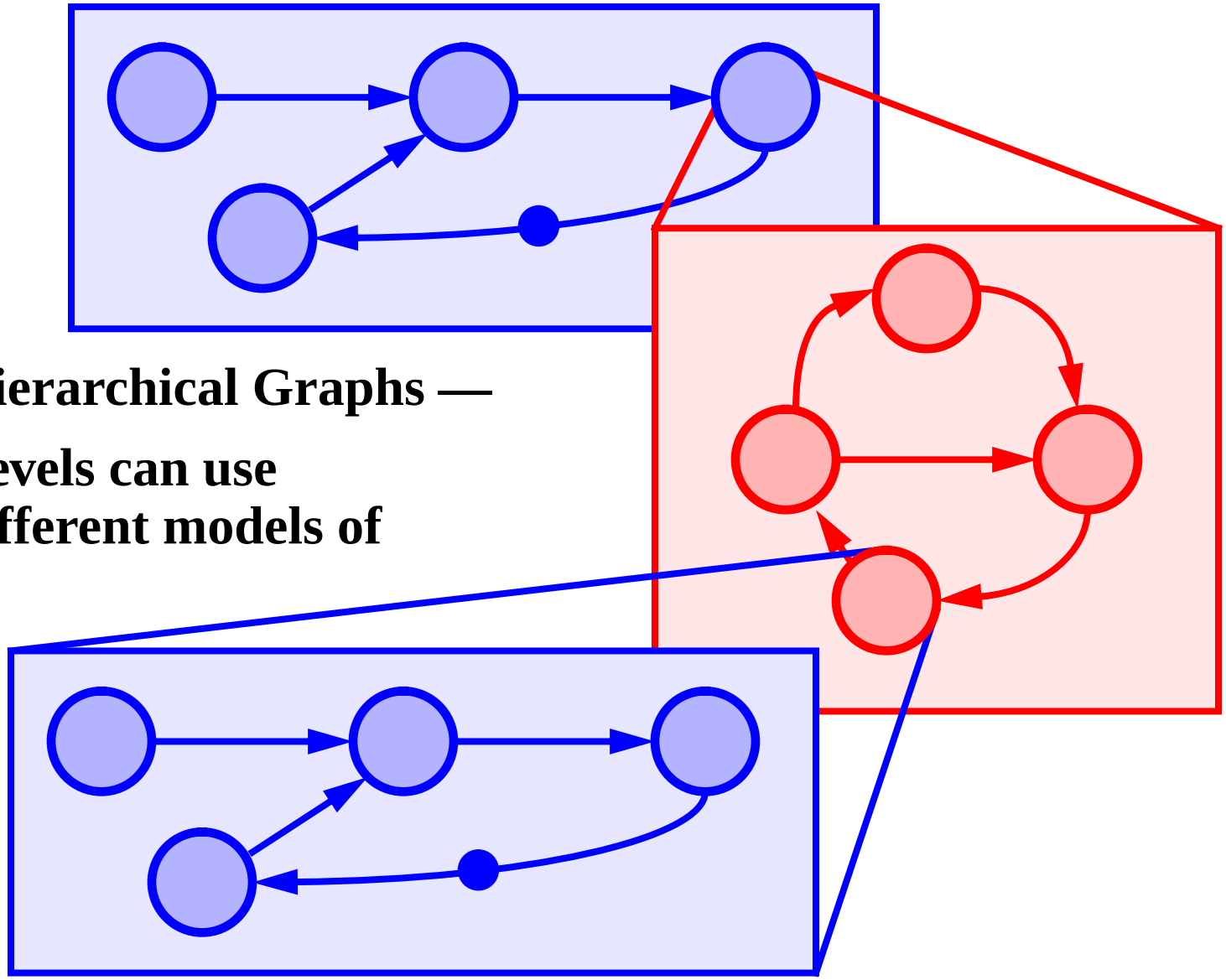
The Almagest

Volume 1
User's Manual
pigi
ptcl
domains
vem
pxgraph
installation

Volume 2
Programmer's Manual
software organization
writing stars
infrastructure
data types
Tcl/Tk
domains
code generation

Volume 3
Kernel Manual
documentation
of C++ classes
Essential for
defining
Targets and
Domains.

The Kernel: A Flexible Abstract Syntax



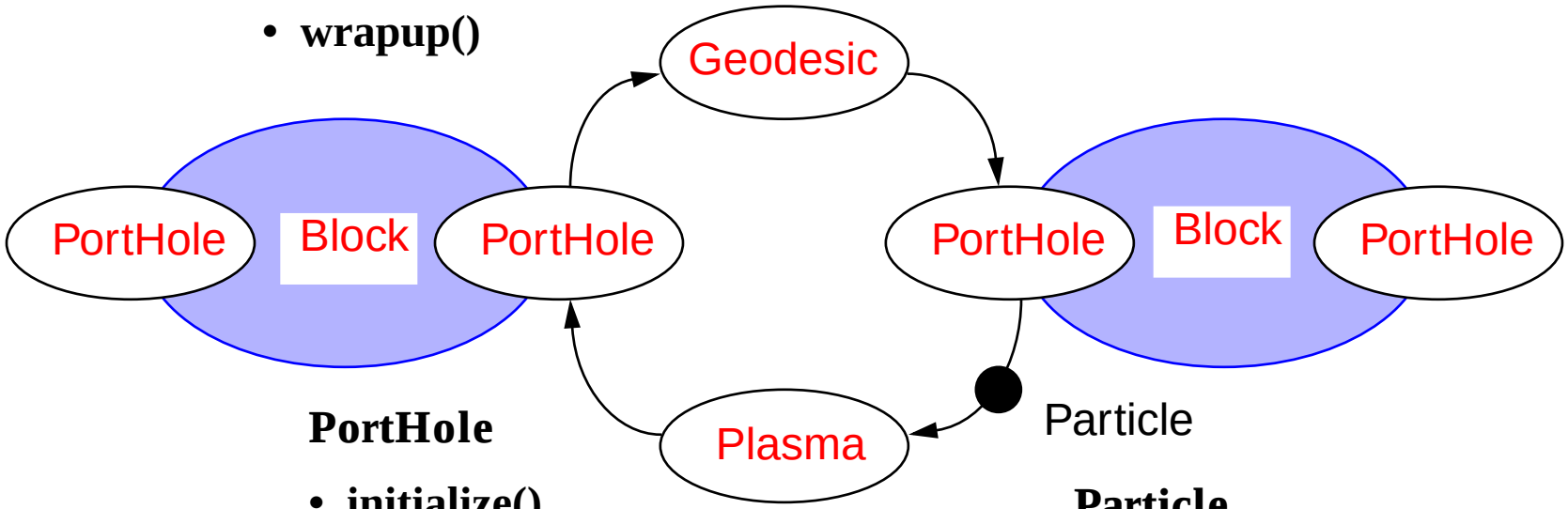
Hierarchical Graphs —

Levels can use different models of

Ptolemy C++ Base Classes Support this Abstract Syntax

Block

- **initialize()**
- **run()**
- **wrapup()**



PortHole

- **initialize()**
- **receiveData()**
- **sendData()**
- **type()**

Particle

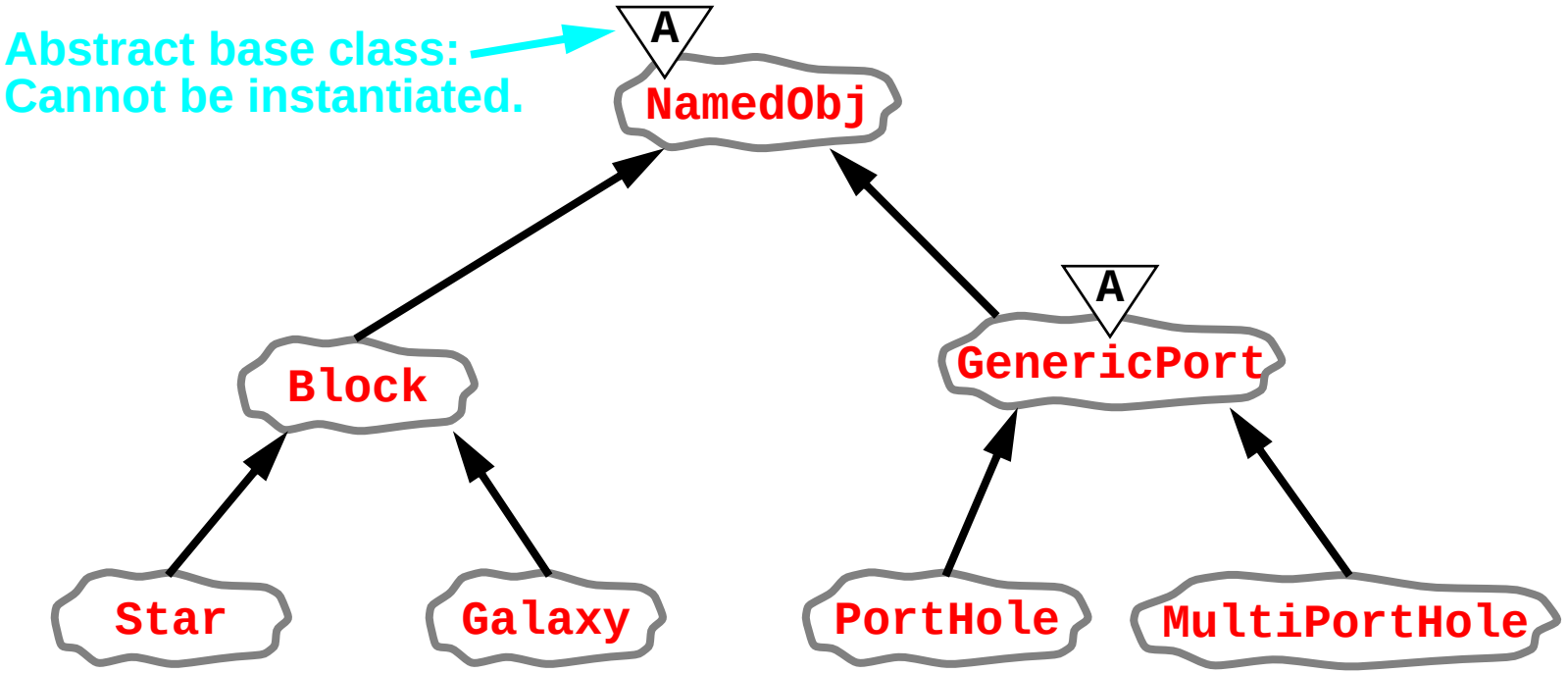
- **type()**
- **print()**
- **initialize()**

Object-Oriented Programming

- **inheritance**
- **data abstraction**
- **polymorphism**

The idea is to use a heterogenous software environment to develop heterogeneous designs. The interaction between different modules in the software environment is managed through object-oriented principles.

Some Kernel Classes (Booch Notation)



C++ class:

- **Members & Methods**
- **Public, Protected, Private**

Inheritance: A derived class has all the properties of the parent, plus more.

Data abstraction: Hiding the irrelevant and amplifying the essential.

Polymorphism: Objects with identical interface have different implementations.

Example: NamedObject

```
class NamedObj {
public:
    NamedObj ();
    NamedObj (const char* n,Block* p,const char* d);

    const char* name() const;
    const char* descriptor() const;
    Block* parent() const;

    void setParent (Block* my_parent);
    void setName (const char* my_name);

    virtual const char* className() const;
    virtual int isA(const char* cname) const;

    virtual void initialize() = 0;
private:
    const char* nm;
    Block* prnt;
    const char* myDescriptor;
}
```

Example: NamedObject

```
class NamedObj {
public:
    NamedObj ();
    NamedObj (const char* n,Block* p,const char* d);

    const char* name() const;
    const char* descriptor() const;
    Block* parent() const;

    void setParent (Block* my_parent);
    void setName (const char* my_name);

    virtual const char* className() const;
    virtual int isA(const char* cname) const;

    virtual void initialize() = 0;
private:
    const char* nm;
    Block* prnt;
    const char* myDescriptor;
}
```

Two possible constructors

Methods to access private members

Methods to set private members

Methods to get class information

Private members

Example: NamedObject

```
class NamedObj {
public:
    NamedObj ();
    NamedObj (const char* n,Block* p,const char* d);
```

The return value cannot be modified.

The string will not be modified

```
    const char* name() const;
    const char* descriptor() const;
    Block* parent() const;

    void setParent (Block* my_parent);
    void setName (const char* my_name);

    virtual const char* className() const;
    virtual int isA(const char* cname) const;
```

Calling a virtual method gives the version in the derived class

```
    virtual void initialize() = 0;
private:
```

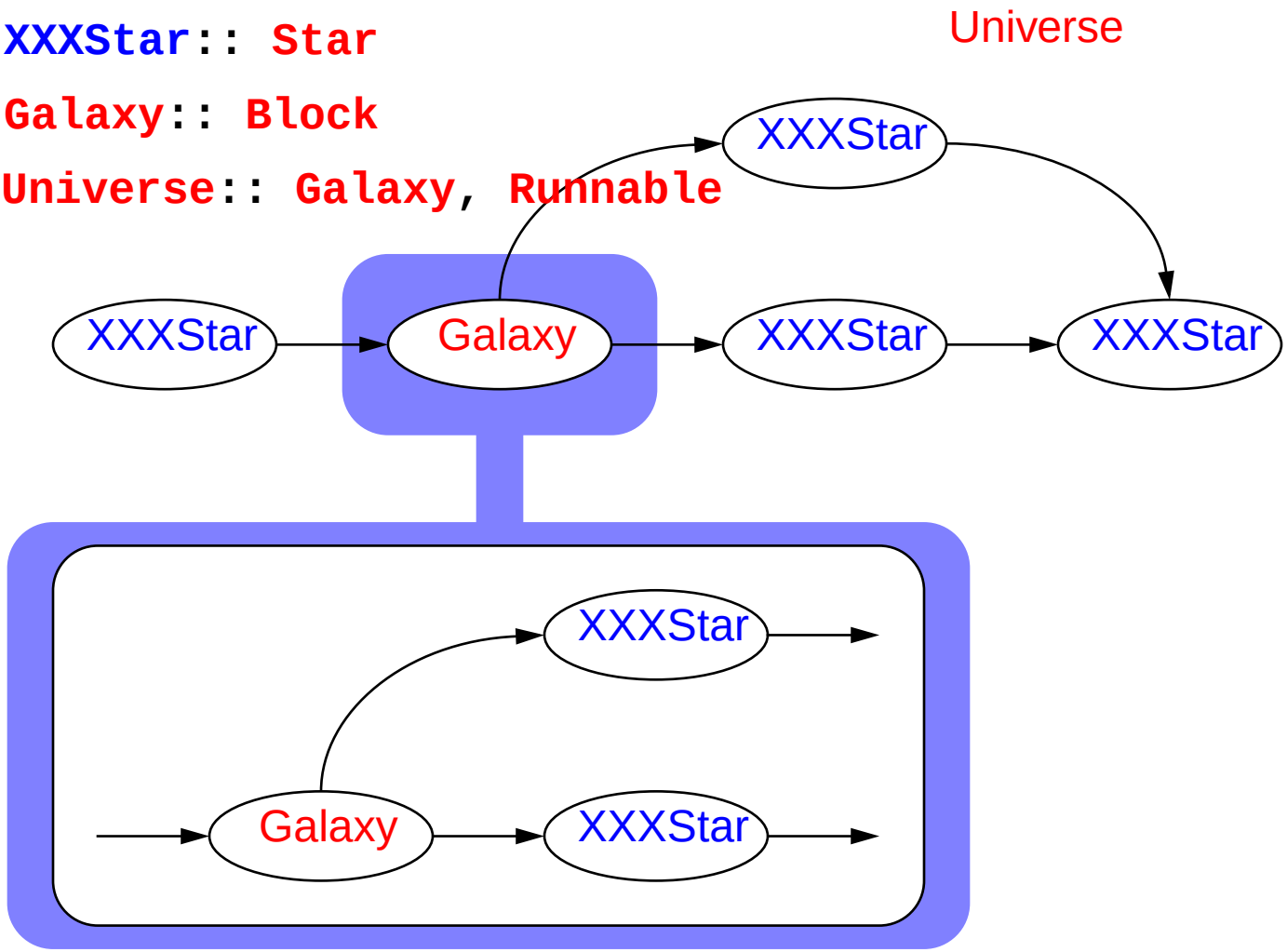
Pure virtual method

```
    const char* nm;
    Block* prnt;
    const char* myDescriptor;
}
```

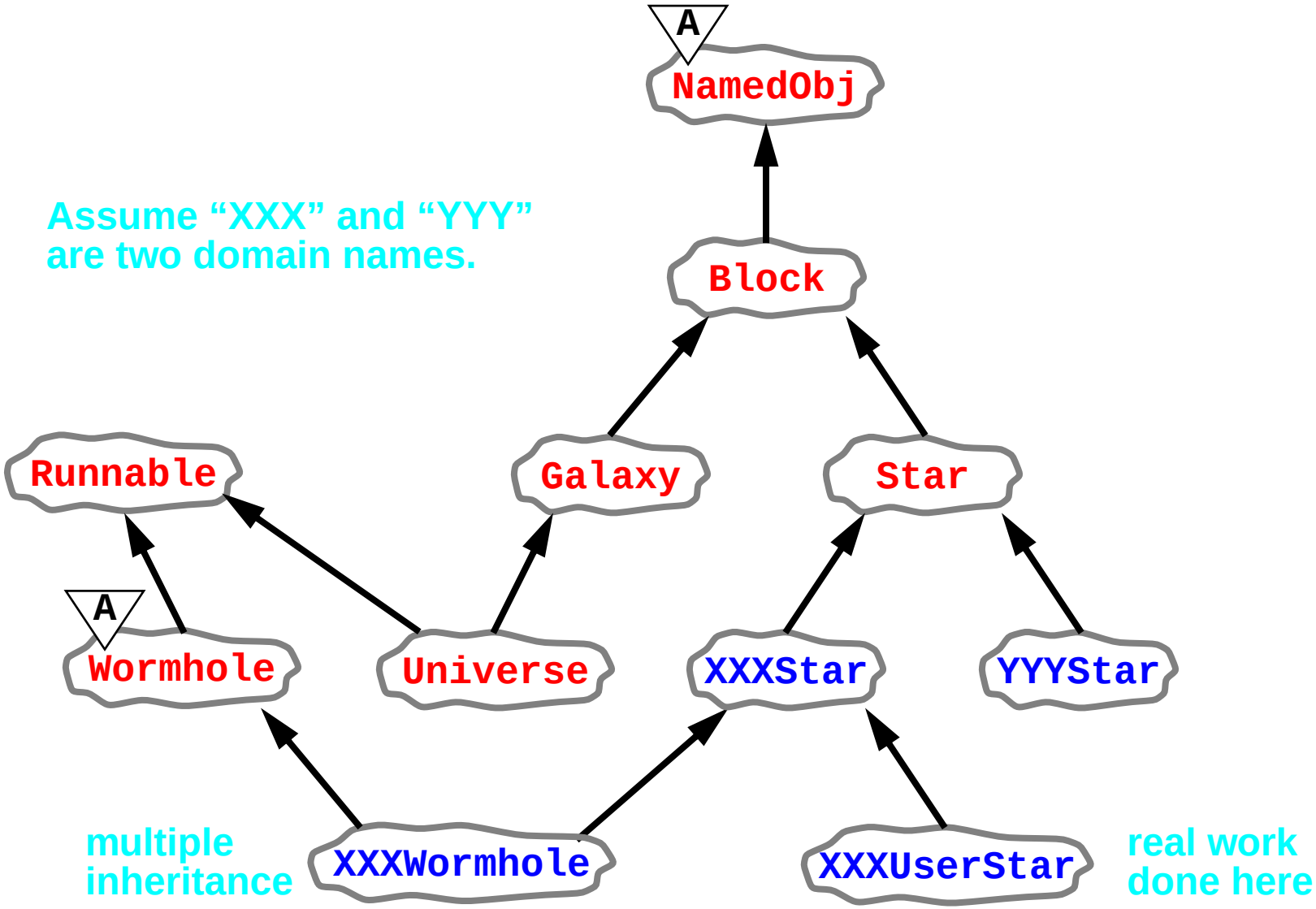
Hierarchical Abstraction

Examples of Derived Classes

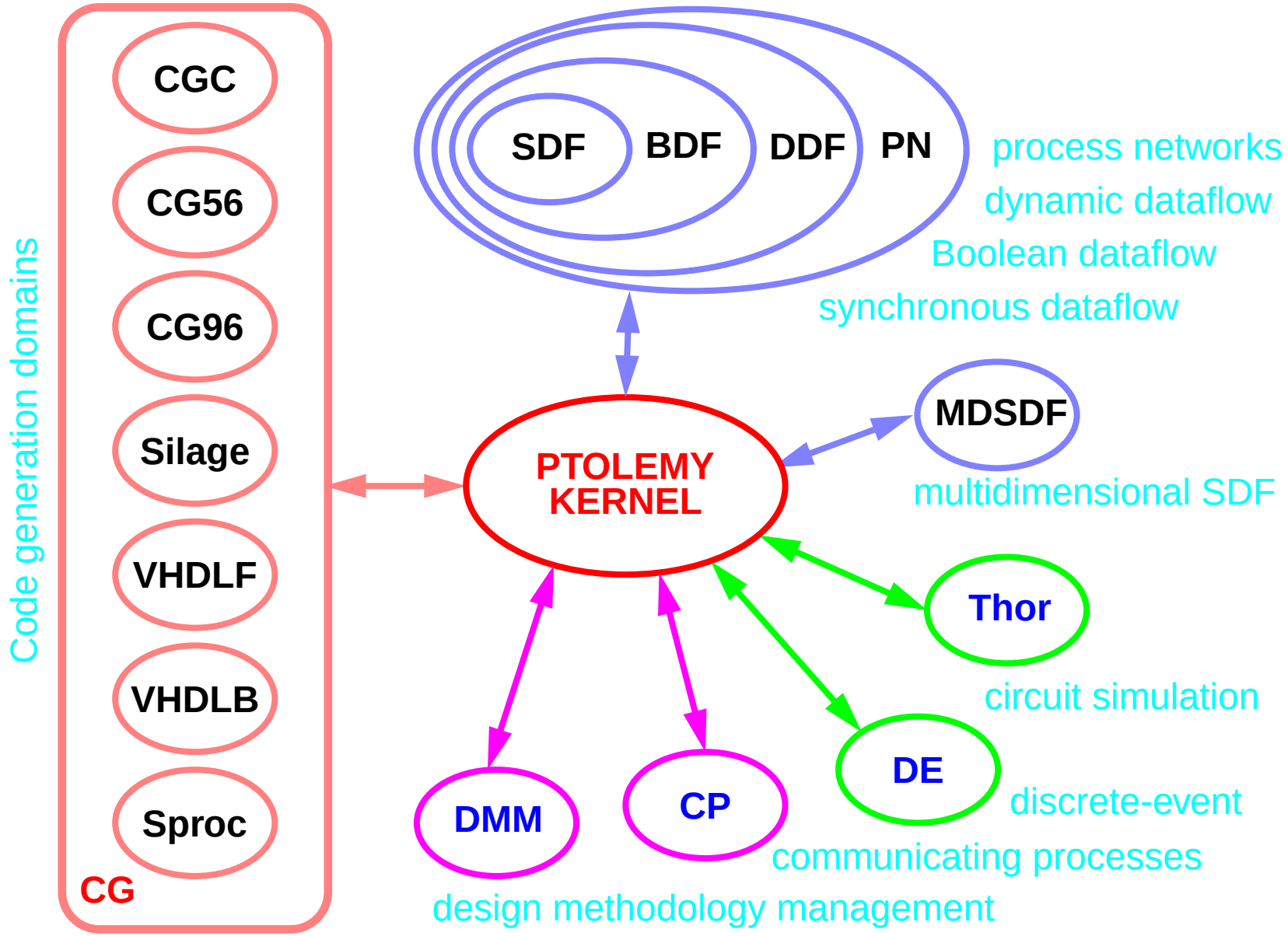
- `class Star:: Block`
- `class XXXStar:: Star`
- `class Galaxy:: Block`
- `class Universe:: Galaxy, Runnable`



Domain Classes Derived from Kernel Classes



Domains in Ptolemy

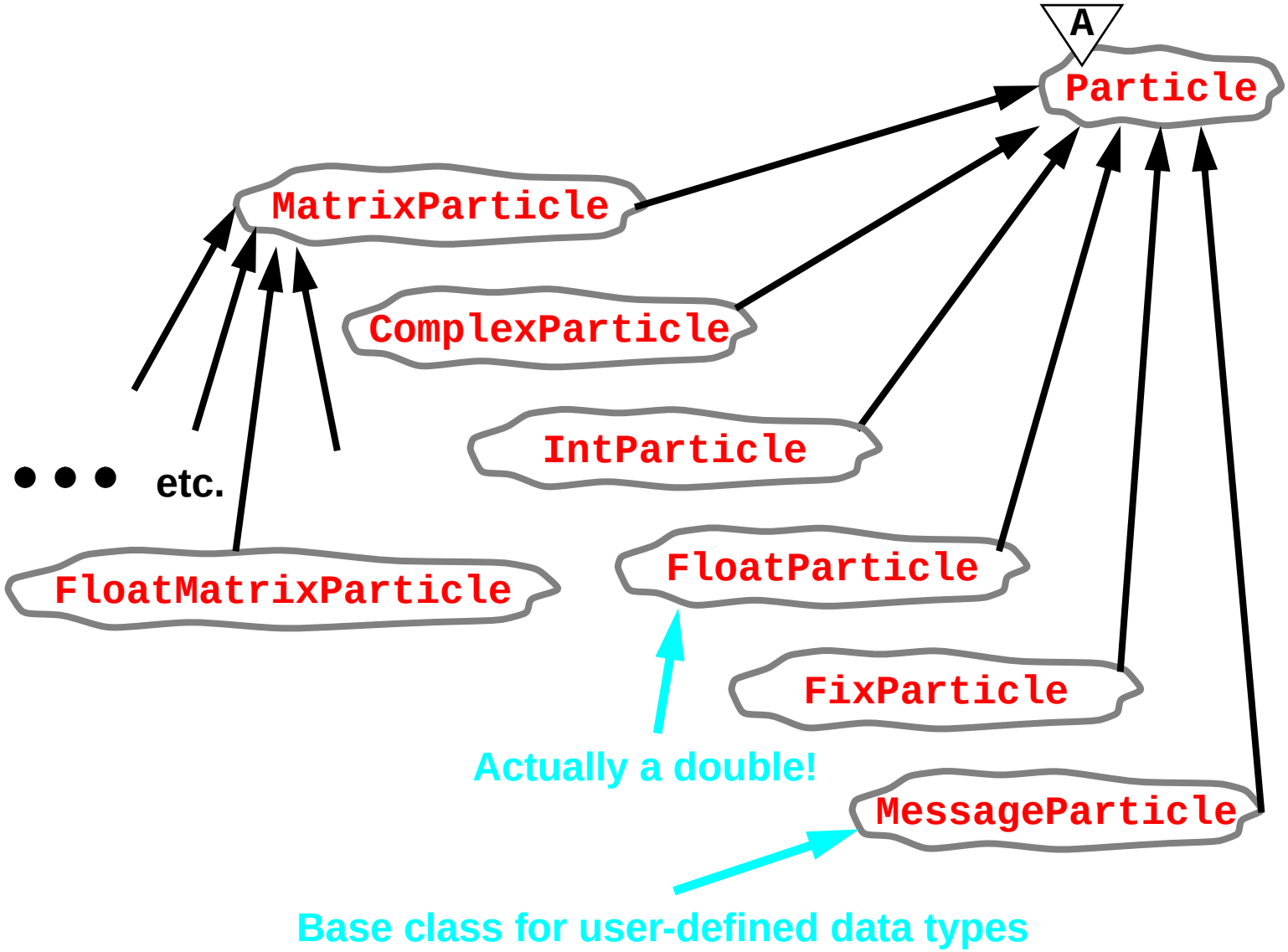


Ptolemy is an Extensible Open Architecture

Extensions that are possible:

- applications (block diagrams)
- a library of galaxies
- application builders (using ptcl or pitcl)
- a library of custom stars
- customized user interfaces (using Tcl/Tk)
- simulation managers (Targets)
- domains (new models of computation)
- foreign simulators and/or synthesis tools

Data Types in Ptolemy



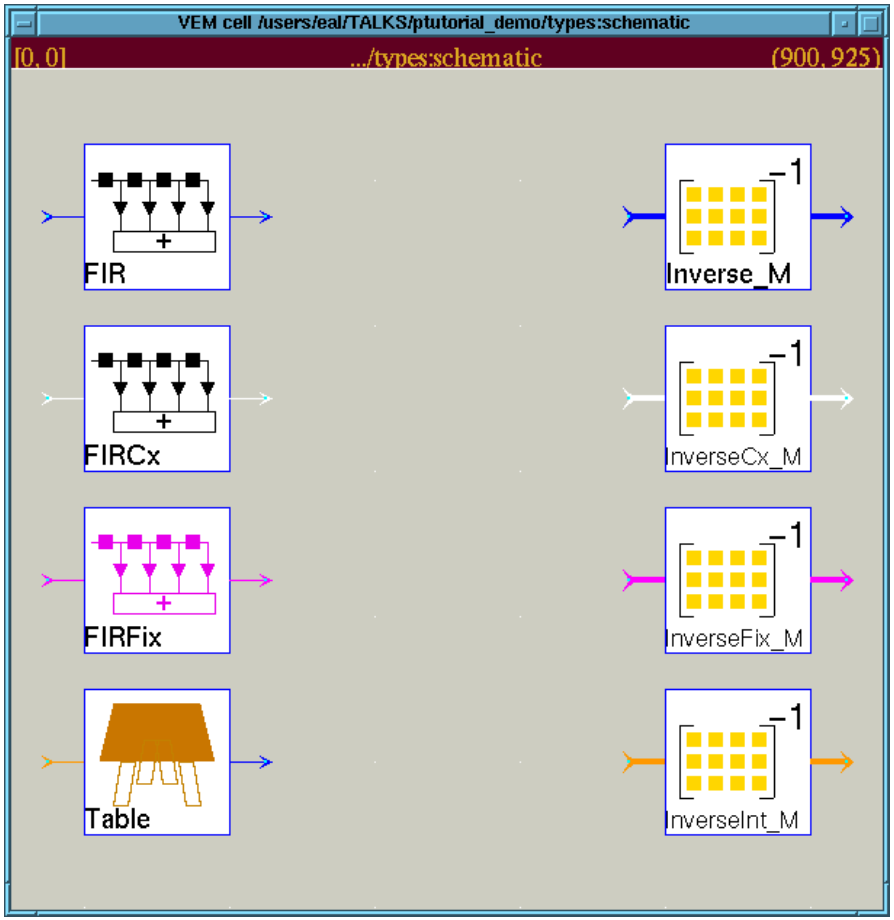
Color Hints at Data Types

blue = float (double)

white = complex

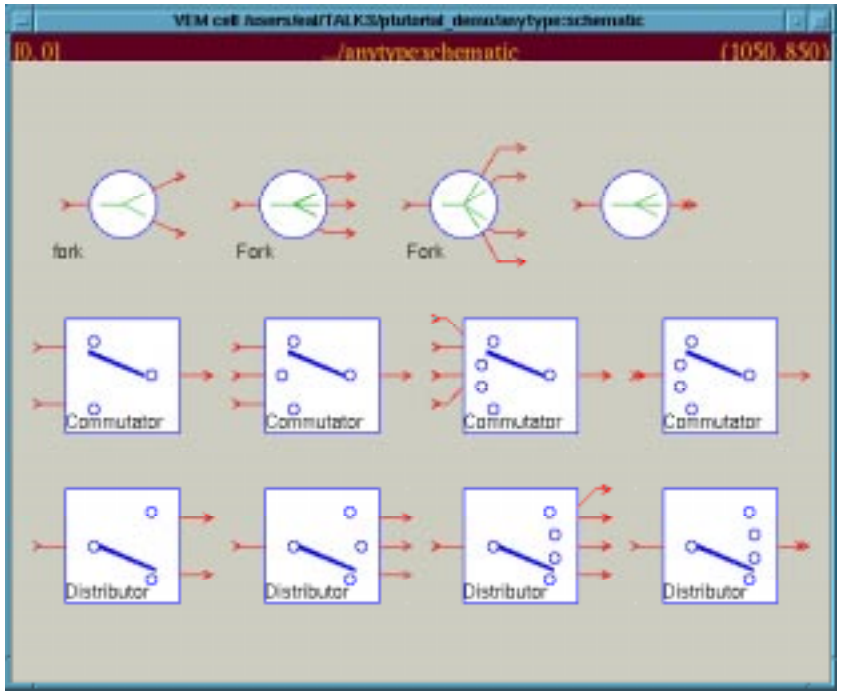
magenta = fixed point

orange = integer



thicker stems indicate matrices

Anytype



Fork:
replicate a particle

Commutator:
merge particle streams

Distributor:
split particle streams

Red stems indicate stars that operate on any particle. The code inside refers only to type "Particle," so these stars are polymorphic.

Key Particle methods

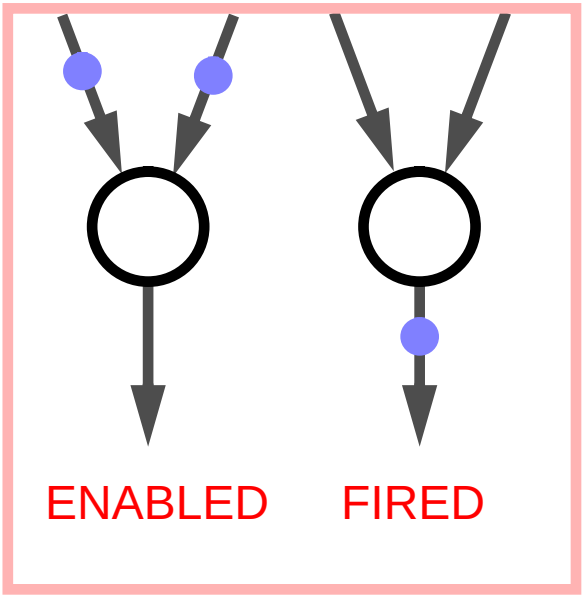
```
class Particle {
public:
    virtual operator int ();
    virtual operator float ();
    virtual operator double ();
    ... etc...

    virtual StringList print ();
    virtual Particle& initialize();

    // Load the Particle with data
    virtual void operator << (int);
    virtual void operator << (double);
    ... etc...

    // Copy a Particle
    virtual Particle* clone();

    // compare two particles
    virtual int operator == (const Particle&);
```

casts to standard types
convert particles to data
and also change types.

generate ASCII

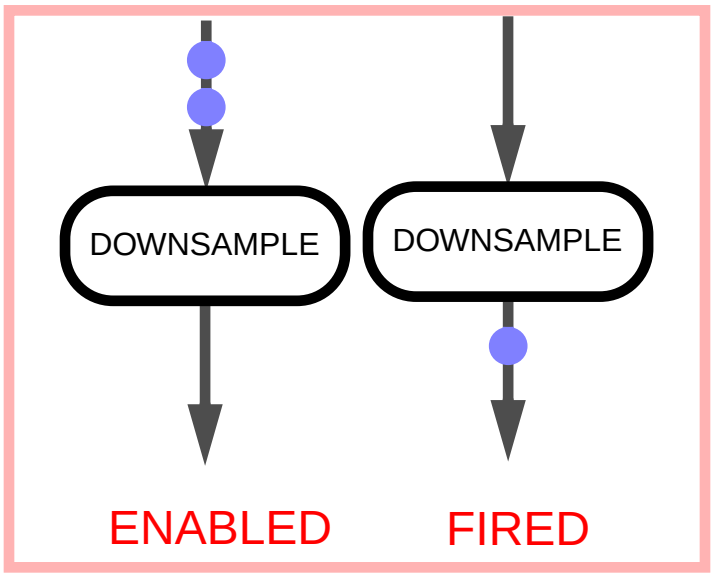
reset to “zero”

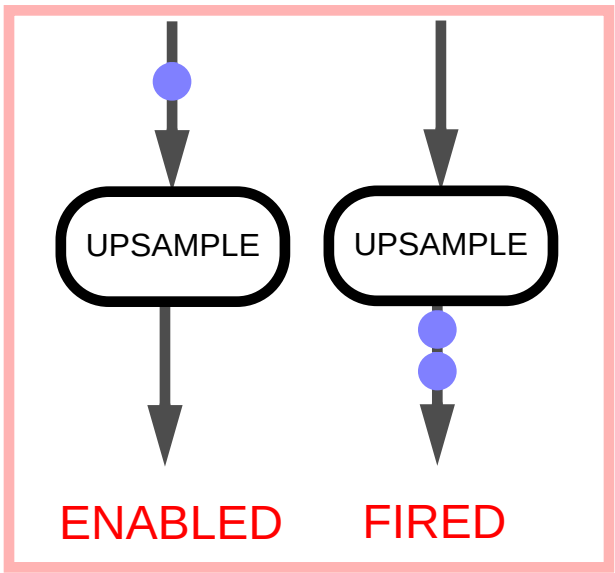
Synchronous Dataflow



Properties

- Flow of control is predictable at compile time
- Schedule can be constructed once and repeatedly executed
- Suitable for synchronous multirate signal processing





Writing Custom Stars

The quick start:

- Find a star in the source tree.
- Copy it and modify it.

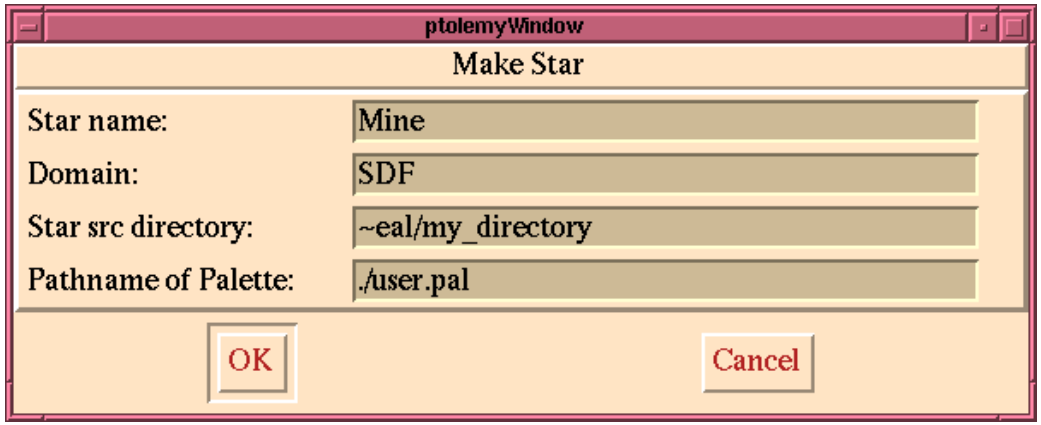
```
cd my_directory
cp $PTOLEMY/src/domains/sdf/stars/SDFSin.pl SDFMine.pl
chmod +w SDFMine.pl
```

- Change the name of the star!
- Replace the appropriate fields with your C++ code.

If your star is a modification of an existing star, consider deriving from the existing star.

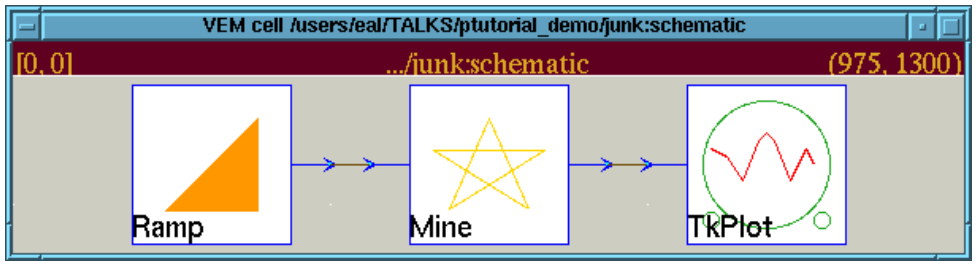
Using Custom Stars

- **Make-star (*)**



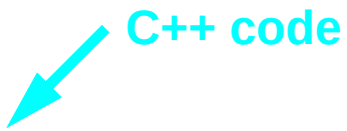
  - **If $PTOLEMY != /users/ptolemy, you have to set four (!) environment variables, or the compiler will not work. This is a flaw with the gnu compilers. See the programmer's manual.**

- **Make a test application**



Minimal Star Definition File

```
defstar {
    name { SimpleLog }
    domain {SDF}
    desc { This star computes the Log of the input }
    input {
        name{input}
        type{float}
    }
    output {
        name{output}
        type{float}
    }
    ccinclude { <math.h> }
    go {
        output%0 << log (double(input%0));
    }
}
```



This should go in a file called “SDFSimpleLog.pl”.

A preprocessor (ptlang) translates it into a C++ class.

Input and Output for Stars

```
go {
    output%0 << log (double(input%0));
}
```

Returns a reference to a Particle

Extracts a double from the Particle

Loads a double into the Particle referenced

Manipulating Particles Directly (for Anytype)

```
go {
    Particle& current = input%0;
    outputA%0 = current;
    outputB%0 = current;
}
```

object PortHole has operator % defined to return a Particle&

More Sophisticated Go Method

```
go {
    double t = double(input%0);
    if (t <= 0) {
        Error::abortRun (*this, ": log of x, x <= 0");
        output%0 << -100.0;
    }
    else output%0 << log(t);
}
```

For most compilers, this explicit cast is not strictly necessary. Still it is safest to use it.

tells the error handler where the error occurred

One of many infrastructure classes provided to star writers.

Accessing Past Samples

```
go {
    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += double(input%i);
    }
    output%0 << sum;
}
```

accessing past samples

need to declare that this will be done:

```
setup {
    input.setSDFParams(1,9);
}
```

second argument tells the domain what the largest argument to the % operator will be. This allows buffers to be allocated statically and old Particles to be reclaimed.

Consuming Multiple Samples

```
go {
    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += double(input%i);
    }
    output%0 << sum;
}
```

the go method is completely unchanged, but:

```
setup {
    input.setSDFParams(10,9);
}
```

the first argument tells the domain how many tokens to discard after the star fires.

Using Parameters

```
defstar {
    name { MyAverage }
    domain {SDF}
    desc { This star averages a batch of inputs }
    input { name{input} type{float} }
    output { name{output} type{float} }
    state { name{howmany} type{int} default{10}}
    setup {
        input.setSDFParams(int(howmany),int(howmany)-1);
    }
    go {
        double sum = 0.0;
        for (int i = 0; i < int(howmany); i++) {
            sum += double(input%i);
        }
        output%0 << sum;
    }
}
```

Fork star

```
defstar {
    name {Fork}
    domain {SDF}
    input {
        name {input}
        type {anytype}
    }
    outmulti {
        name {output}
        type {=input}
    }
    go {
        MPHIter iterator(output);
        PortHole* p;
        while ((p = iterator++) != 0) {
            (*p)%0 = input%0;
        }
    }
}
```

declares the star to be polymorphic

output will be a MultiPortHole object

note type propagation

iterators are a basic mechanism for stepping through lists. They have operator ++ that returns the next object on the list, and returns 0 when the list is finished.

Methods in a Star

Standard methods:

- **constructor: called when the star object is created**
- **setup: called before the scheduler is initialized**
- **begin: called after the scheduler is initialized, before the run**
- **go: called during the run**
- **wrapup: called at the completion of an *error-free* run**
- **destructor: called when the star object is destroyed**

You can also include in a star any other methods of your own design, plus arbitrary code segments in C or C++.

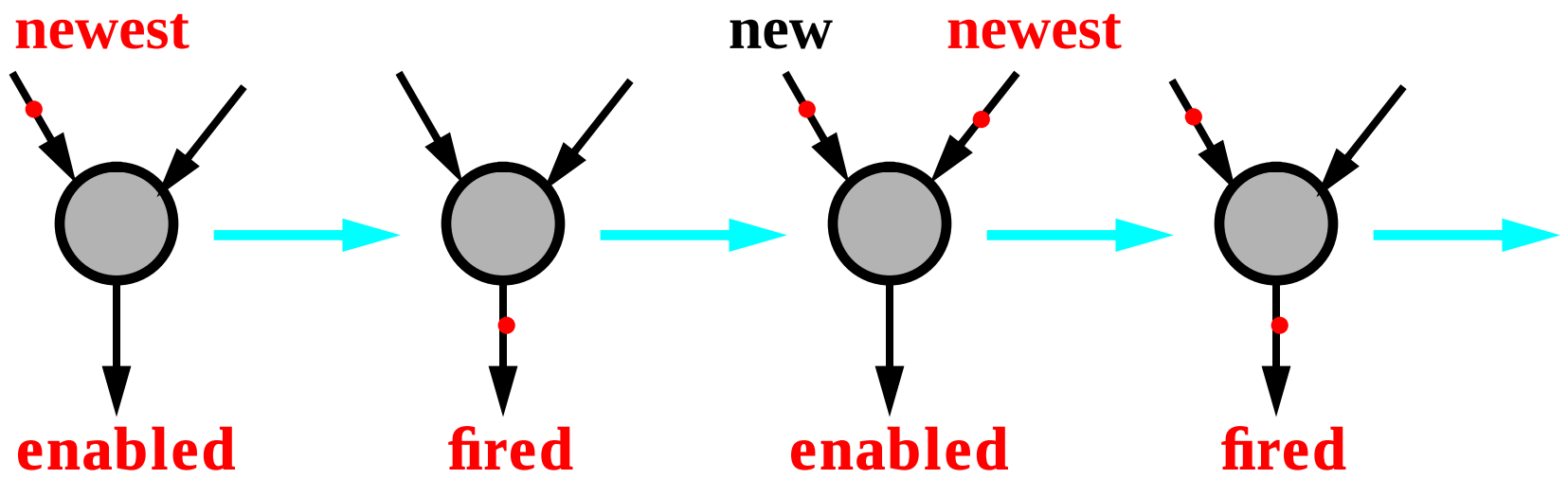
Notes and Extensions

- By convention, we call *states* that do not change in value during an execution *parameters*, although Ptolemy treats them the same.
- There are array states, string states, and string array states, as well as all the standard types.
- Calling Error::abortRun does not immediately stop a run. It just alerts the scheduler to stop at a convenient time.
- Other infrastructure classes include:
  - I/O stream classes
  - String classes
  - XGraph
  - BarGraph
  - List classes
  - Hash tables
  - Complex class
  - Fix class
  - Matrix classes
  - Random numbers
  - Histogram

Discrete-Event Domain

Discrete-Event Domain:

- A *signal* is a sequence of *discrete events* (data + timestamp).
- Discrete events are ordered in a global *queue* by timestamp.
- The global queue schedules which *actor* to execute.
- An actor *consumes* input tokens, *produces* output tokens.



Execution Model for the Discrete-Event Domain

Run-Time Scheduling

- Global event queue sorts events by their timestamps
- Scheduler fetches the event at the head of the queue and sends it to the input ports of its destination block

Special Cases

- Initialization: insert triggering particles in the event queue.
- Simultaneous events at all input ports to an actor are provided when the actor is executed
- Equal timestamps in the event queue are sorted according to a topological sort:
  1. partial ordering of the actors in the flow graph,
  2. triggering relationships between the input and output ports, and
  3. ordering of the input ports.

Discrete-Event Actors and Channels

Actors

- **Functional actors**
  timestamp of data on output ports =
  timestamp of data on triggering input ports
- **Delay actors**
  timestamp of data on output ports =
  timestamp of data on triggering input ports + ***delay***

Properties of Channels

- **Delay attribute**
  represents an infinitesimal delay (sort of)
  introduces ***no*** **actual delay in simulated time**
  breaks topological ordering between connected actors
  prevents race conditions
  suppresses delay-free loop warnings

Delay Star Definition

```
defstar {
    name { Delay }
    domain {DE}
    desc { Delays its input by a fixed amount }
    input {
        name {input}
        type {anytype}
    }
    output {
        name {output}
        type {=input}
    }
    defstate {
        name {delay}
        type {float}
        default {"1.0"}
        desc { Amount of time delay. }
    }
```

inherits from DEStar class

Single-input, single-output block

in simulated time units

File is "DEDelay.pl" in $PTOLEMY/src/domains/de/stars

Continued on the next page

Delay Star Definition

```
constructor {
    delayType = TRUE;
}
```

Member of the DEStar class

The star is a delay star

arrivalTime (member of the DEStar class) is set by the scheduler to the timestamp of the event(s) the triggered the firing of the star

Manipulating Timestamps

```
go {
    completionTime = arrivalTime + double(delay);
    Particle& pp = input.get();
    output.put(completionTime) = pp;
}
```

completionTime (member of DEStar class) records latest output time

More Sophisticated Delay Star (a Server)

Input event waits until a simulated resource becomes free to attend to it

Resource is free

```
go {
    if ( arrivalTime > completionTime )
        completionTime = arrivalTime + double(delay);
    else
        completionTime += double(delay);
    Particle& pp = input.get();
    output.put(completionTime) = pp;
}
```

delay parameter indicates the service time

Server Star

Functional Star

```
defstar {
    name { Switch }
    domain {DE}
    desc { Switches input events to one of two outputs,
depending on the last received control input. }
    input {
        name { input }
        type { anytype }
    }
    input {
        name { control }
        type { int }
    }
    output {
        name { true }
        type { =input }
    }
    output {
        name { false }
        type { =input }
    }
```

Functional Stars

```
constructor {
    control.triggers();
    control.before(input);
}
```

Control input does not trigger outputs

For equal timestamps, read control input first

Checking for New Data on Data Input

```
go {
    if ( input.dataNew ) {
        completionTime = arrivalTime;
        Particle& pp = input.get();
        int c = int(control%0);
        if ( c )
            true.put(completionTime) = pp;
        else
            false.put(completionTime) = pp;
    }
}
```

Extending the GUI

Definitions:

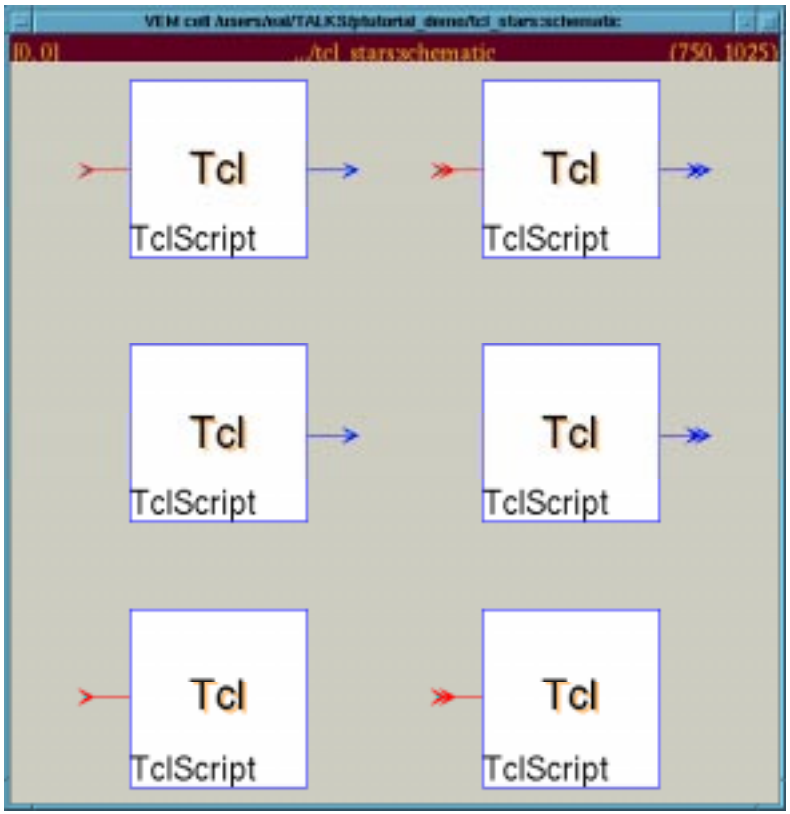
- **Tcl is an interpreted "tool command language" designed by John Ousterhout at Berkeley.**
- **Tk is an associated X window toolkit.**

Essential reference:

J. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, Reading, Mass., 1994.

Tcl and Tk scripts may be invoked by stars and targets.

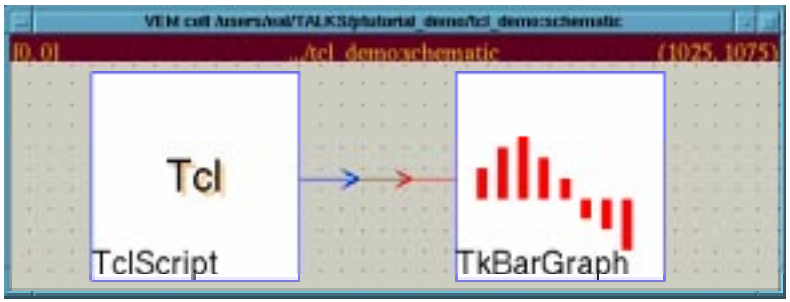
TclScript stars



The single parameter is the name of a Tcl file that is sourced at setup time. The behavior of the star is determined by this file.

Example

```
# Create an empty top-level window
toplevel .top

# Put a button in the window
pack [button .top.b -text “PUSH ME”]

# Bind an action to pushing the button
bind .top.b <ButtonPress-1> “setOutputs_$starID 1.0”
bind .top.b <ButtonRelease-1> “setOutputs_$starID 0.0”
```



Notes:

- you cannot use more than one instance of this script (name conflict).
- this script fails to remove the window it creates.
- the system runs free, unsynchronized to the button.

An Awkward way to be Quasi-Object-Oriented

Global variables that are set before your script is sourced:

- **starID (a unique identifier for your star)**
- **ptkControlPanel (the name of the control panel window)**

Star procedures you can call from your script

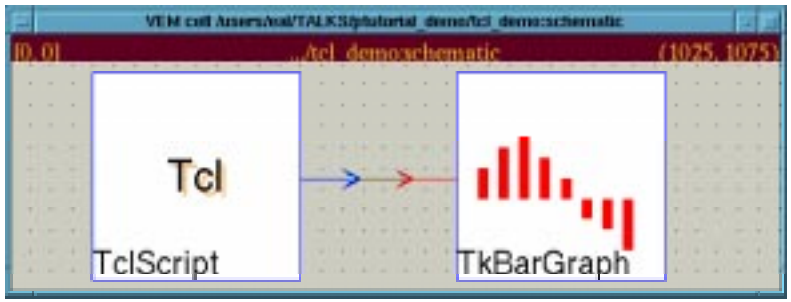
- **setOutputs_$starID**
- **grabInputs_$starID**

Procedures you can define in your script

- **goTcl_$starID**
- **wrapupTcl_$starID**
- **destructorTcl_$starID**

Putting the Button in the Control Panel

```
# Use a variable to store the long window name
set s $ptkControlPanel.middle.button_$starID

# Put a button in the window
pack [button $s -text “PUSH ME”]

# Bind an action to pushing the button
bind $s <ButtonPress-1> “setOutputs_$starID 1.0”
bind $s <ButtonRelease-1> “setOutputs_$starID 0.0”
```

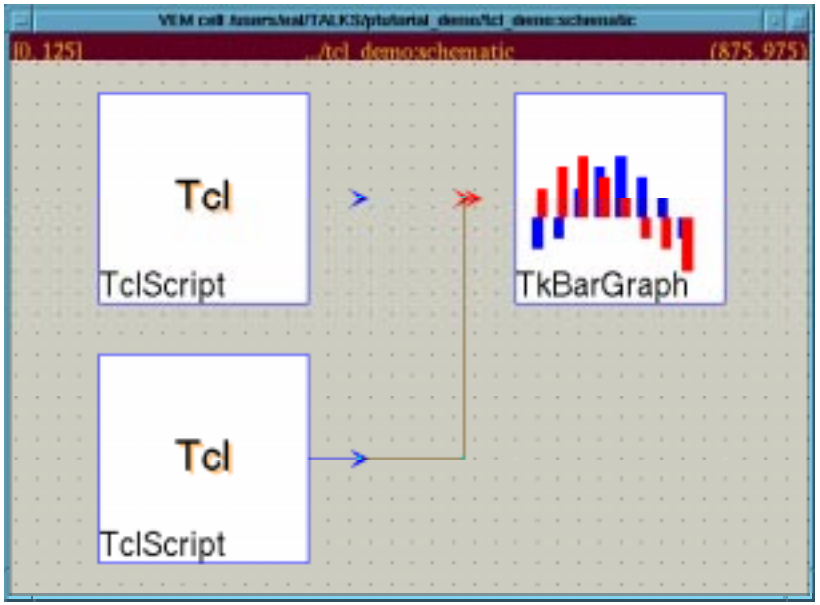


Notes:

- no more name conflict (try using more than one instance).

Allowing Repeated Runs

```
set s $ptkControlPanel.middle.button_$starID

# Add the buttons only if they don't already exist
if {! [winfo exists $s]} {
    pack [button $s -text "PUSH ME"]
    bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
    bind $s <ButtonRelease-1> "setOutputs_$starID 0.0"
}
unset s
```



- **buttons are only created if they don't already exist.**

Parameters

A global array with the name given by the value of the starID is created to store the following information:

- **[set ${starID}(numInputs)]: The number of inputs actually connected to the star.**
- **[set ${starID}(numOutputs)]: The number of outputs actually connected to the star.**
- **[set ${starID}(tcl_file)]: The name of the Tcl file used by the star.**

In addition, a set of procedures are provided for convenience and for uniformity of appearance:

- **ptkExpanEnvVar**
- **ptkImportantMessage**
- **ptkMakeButton**
- **ptkMakeEntry**
- **ptkMakeMeter**
- **ptkMakeScale**

Another Example

```
# This procedure is invoked every time the star fires,
# thus providing a synchronized run
proc goTcl_$starID {starID} {

    # Suppose we know that the star has two inputs.
    # We can read them and add them together as follows.
    set inputVals [grabInputs_$starID]

    # Split the input list into individual elements
    set xin [lindex $inputVals 0]
    set yin [lindex $inputVals 1]

    # Add the numbers and send to the output
    setOutputs_$starID [expr $xin+$yin]
}
```

Note that Tcl is not a good way to do arithmetic, in general. Too slow. Also, the above star should iterate over however many inputs are actually connected.

Creating Custom Stars Derived from TclScript

```
defstar {
    name {MyFancyWidgetStar}
    derivedFrom {TclScript}
```

Add your own parameters:

```
    state {
        name{howManyWidgets}
        type{int}
        default{10}
    }
```

Hide the tcl_file parameter:

```
    setup {
        tcl_file = "~me/my_directory/myfile.tcl";
        tcl_file.clearAttributes(A_SETTABLE);
    }
```

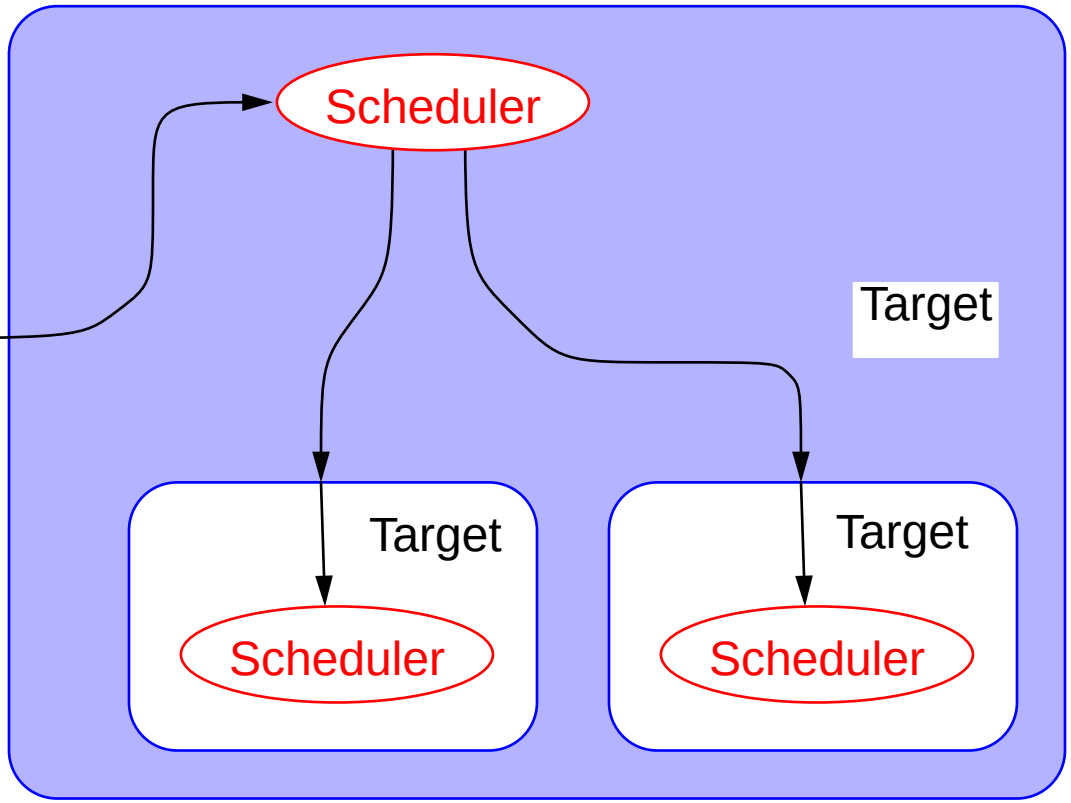
Your parameters are accessible in your Tcl script:

```
    set n [set ${starID}(howManyWidgets)]
```

Managing Simulation or Code Generation

Target:: Block

- initialize()
- setup()
- run()
- wrapup()
- galaxy
- scheduler
- children



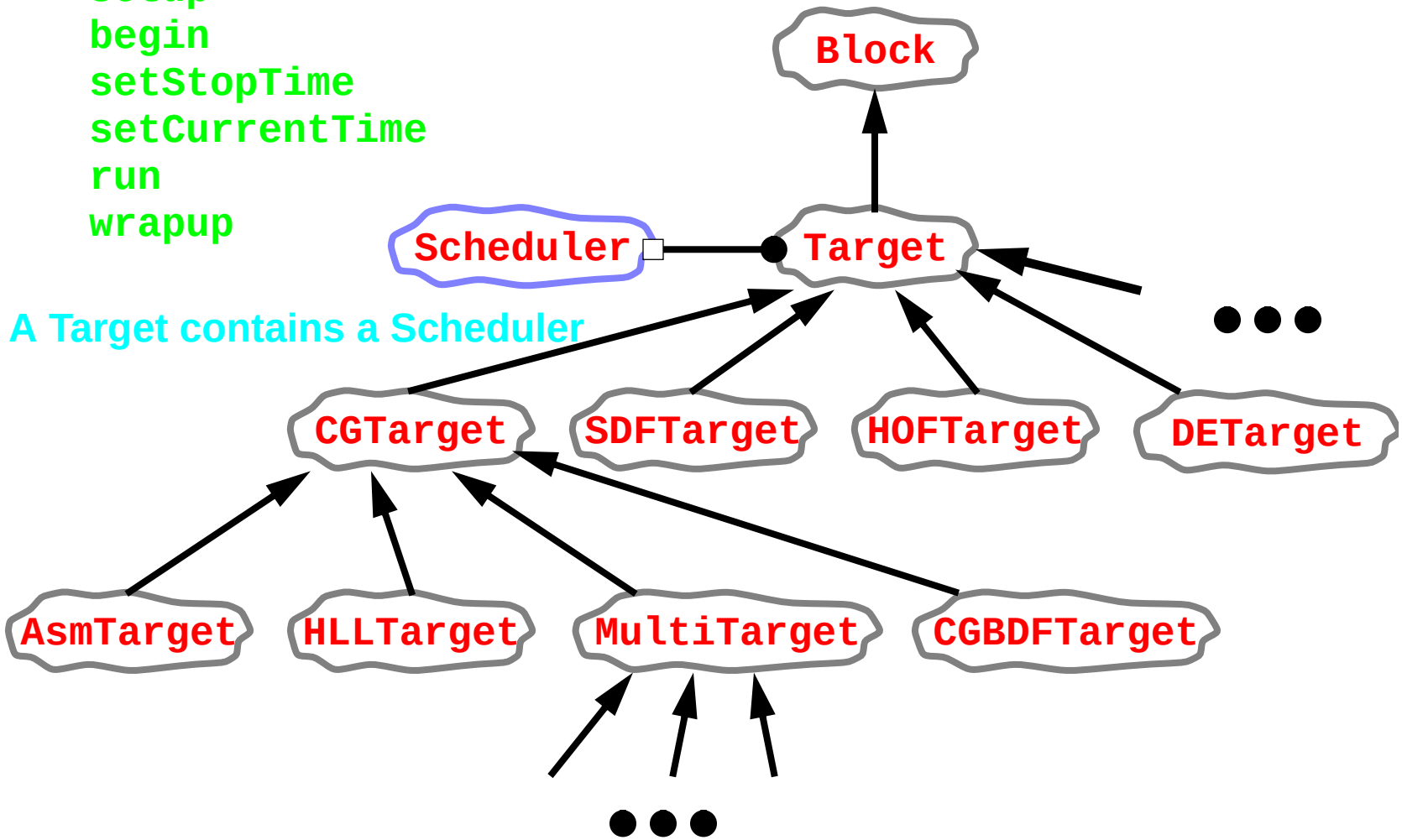
Defining Targets

- A Target supervises the execution of Universe
- A Universe has a Target that manages its execution
- A Target normally contains a Scheduler
- The Target is a Block, and hence can have States
- A Target can do simulation, code generation, or compilation
- A Target can have child Targets (e.g. for multiprocessors)

Target Class Structure

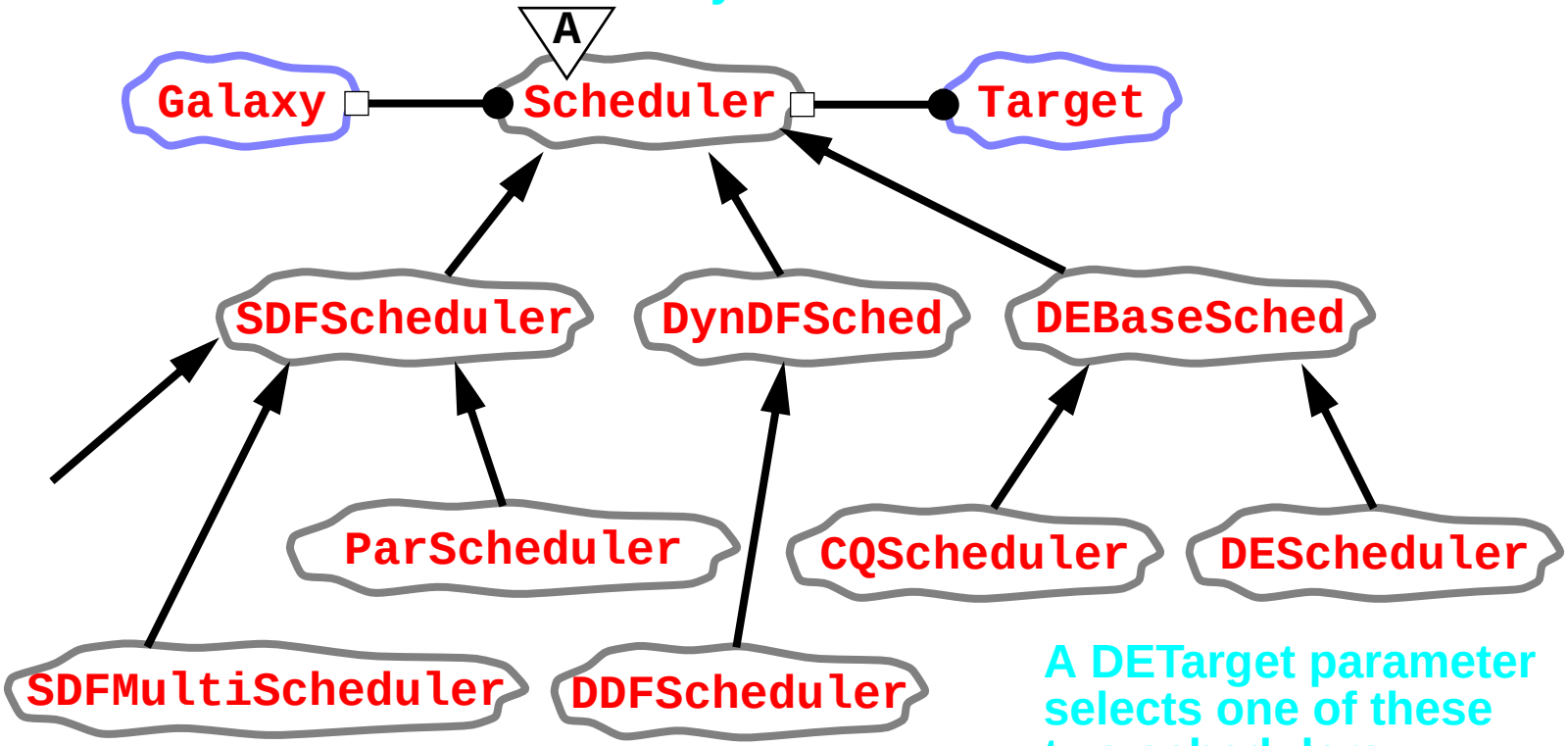
Target methods

```
setup
begin
setStopTime
setCurrentTime
run
wrapup
```



Scheduler Class Hierarchy

A Scheduler contains a Galaxy



A DETarget parameter selects one of these two schedulers

Scheduler methods

```
setup
setStopTime
setCurrentTime
run
```

The World's Simplest Target — Class Definition

```
// The base class template file must be included
#include "Target.h"

class MyTarget : public Target {
public:
  void setup();
  int run();
  void wrapup ();

  // The constructor simply passes the arguments on
  // to the base class constructor.
  MyTarget(const char* nam,const char* stype
       const char* desc) : Target(nam,stype,desc) {}

  // Every Target has a method to make another instance
  // of the same type of Target.
  Block* makeNew() const;
};
```

The World's Simplest Target — Method Definitions

```
// These methods just use the Error class to issue messages

void MyTarget::setup() {
  Error::message("Requesting setup\n");
}

int MyTarget::run() {
  Error::message("Requesting run\n");
  return 1;
}

void MyTarget::wrapup() {
  Error::message("Requesting wrapup\n");
}

// The method to return a new instance
Block* MyTarget::makeNew() const {
  return new MyTarget(name(), starType(), descriptor());
}
```

The World's Simplest Target — Instantiation

```
// A class that keeps track of all available Targets:
#include "KnownTarget.h"

// Create an instance of the Target (there has to be
// at least one instance from which clones are made).
// The arguments are:
//     Target name
//     types of stars supported
//     a short description of the Target
static MyTarget myTargetPrototype("MyTarget", "SDFStar",
           "The world's simplest Target");

// Register the Target in the list of known Targets.

static KnownTarget entry(myTargetPrototype,"MyTarget");
```

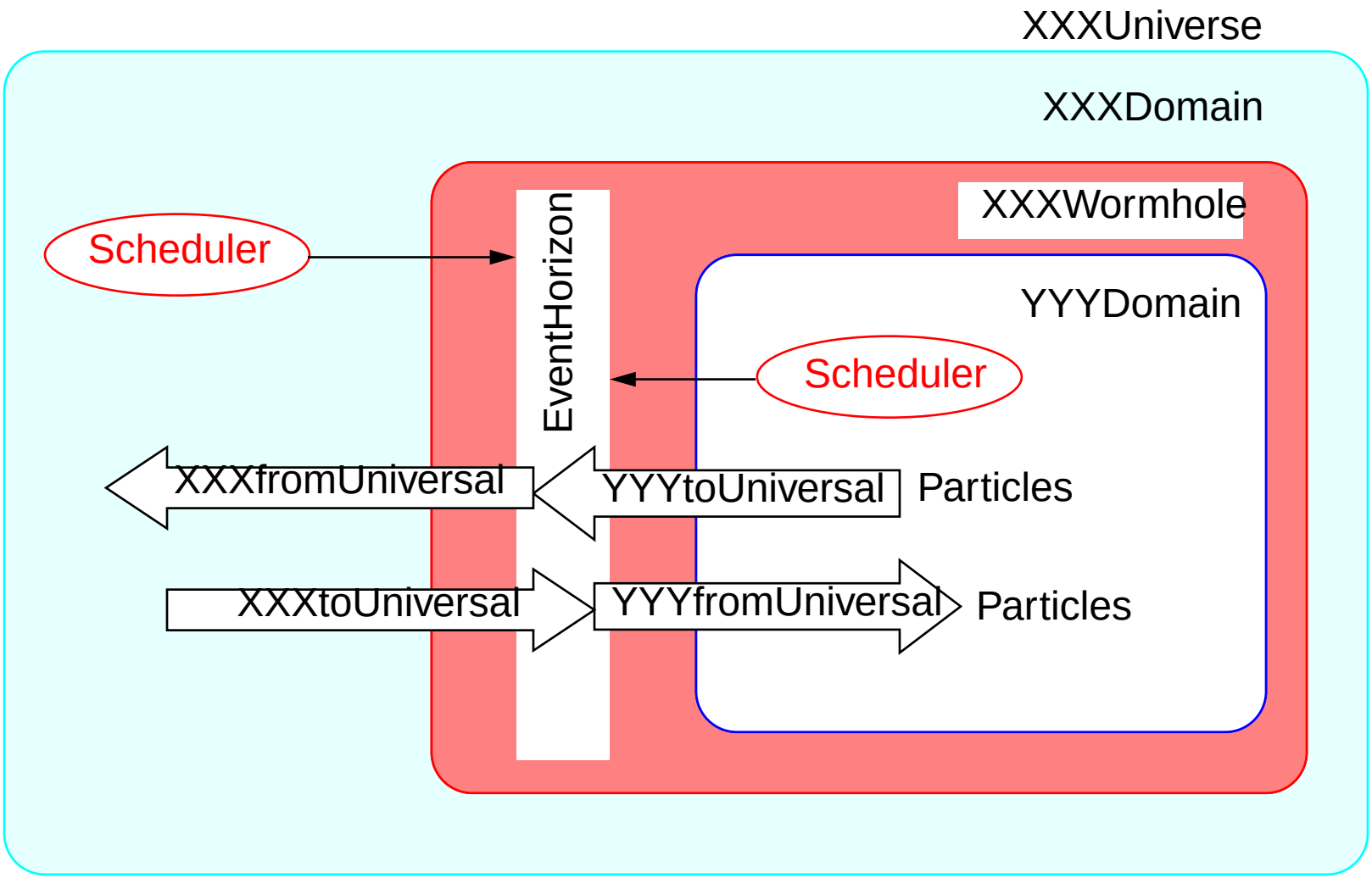
The World's Simplest Target — Compilation

```
# The following extremely simple makefile will work

MyTarget.o:MyTarget.cc
    g++ -fPIC -I$(PTOLEMY)/src/kernel -c MyTarget.cc
```

Slightly More Interesting Target

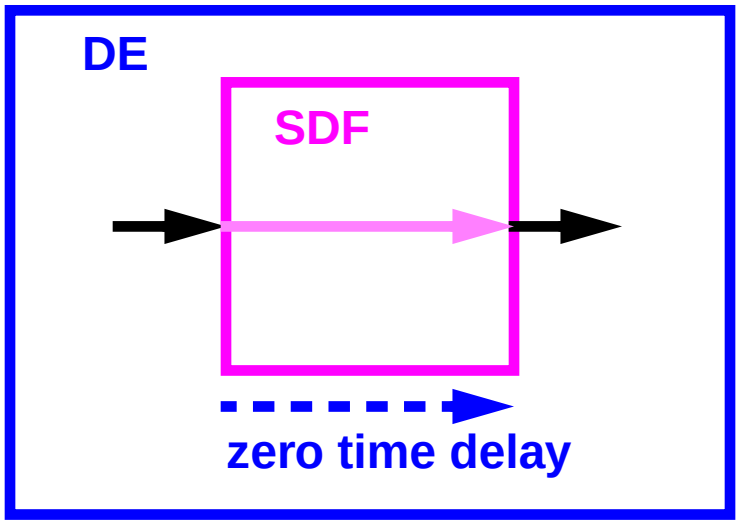
```
// Add the following include files
#include “InfString.h”
#include “GalIter.h”

// Modify the setup method
void MyTarget::setup() {

  // An infinite string class
  InfString list;

  // Construct a list of the blocks at the top level
  // using an iterator.
  GalTopBlockIter blockIterator(*galaxy());
  Block *b;
  while ((b = blockIterator++) != 0) {
    list << b->name() << “\n”;
  }

  // Display the names
  Error::message(list);
}
```

Interaction Across Models of Computation



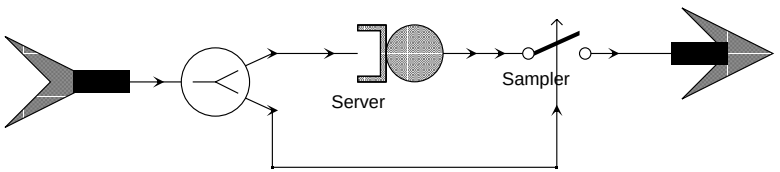
SDF Domain Inside a DE Domain

- SDF domain is treated as a functional DE star (zero delay)
- Multirate SDF subsystems may not always produce output for a single input



DE Subsystem Inside an SDF Subsystem

- SDF subsystem treats the DE subsystem as an SDF star: it must consume and produce a fixed number of tokens
- SDF subsystem will expect the DE subsystem to produce an output when given input



The DE sampler star ensures that the appropriate number of output events are produced in response to input events.

When Should you Define a New Domain

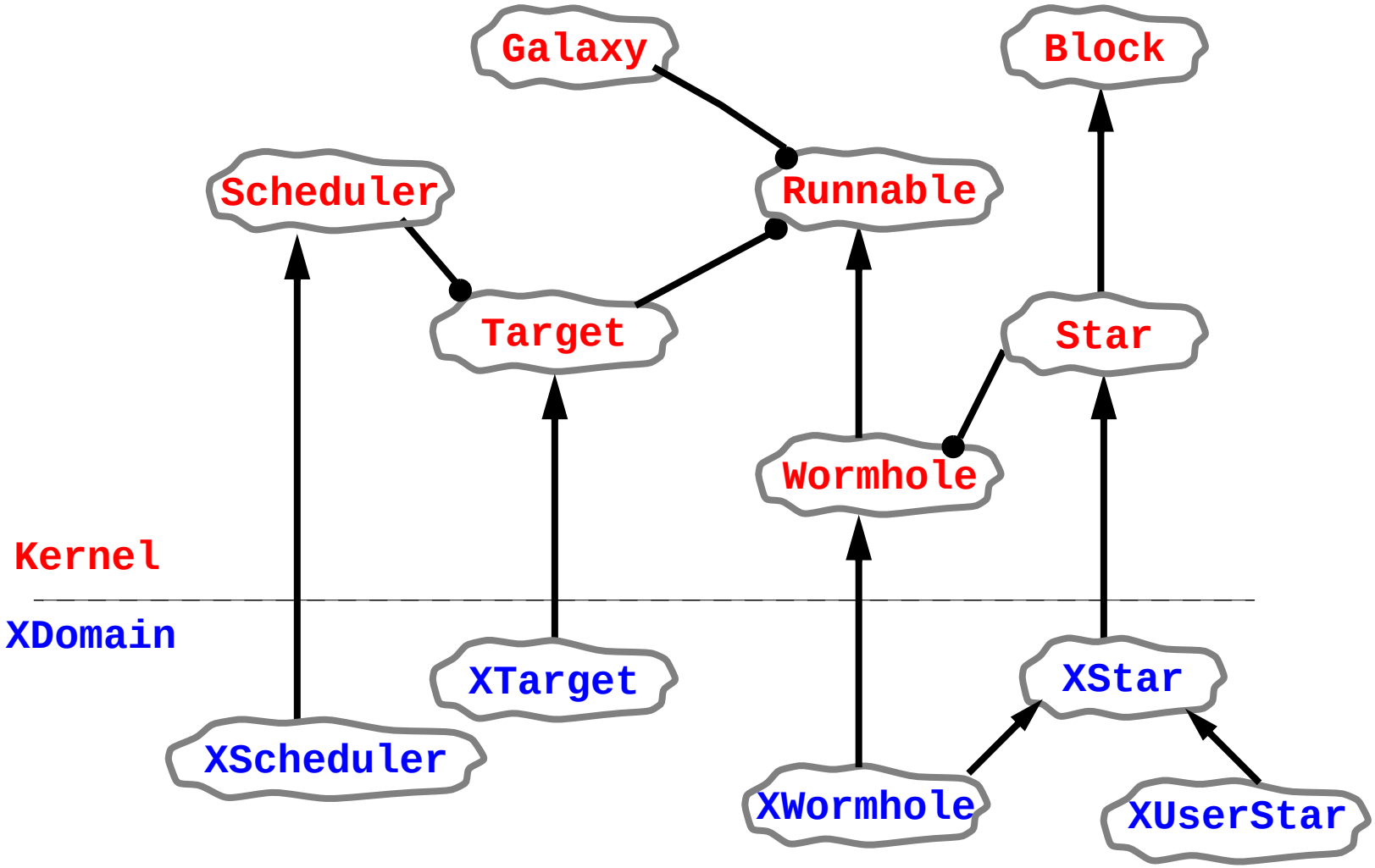
All of the following should be true:

- You need a new model of computation, X.
- You understand all existing domains.
- The stars in each existing domain are incompatible with X.

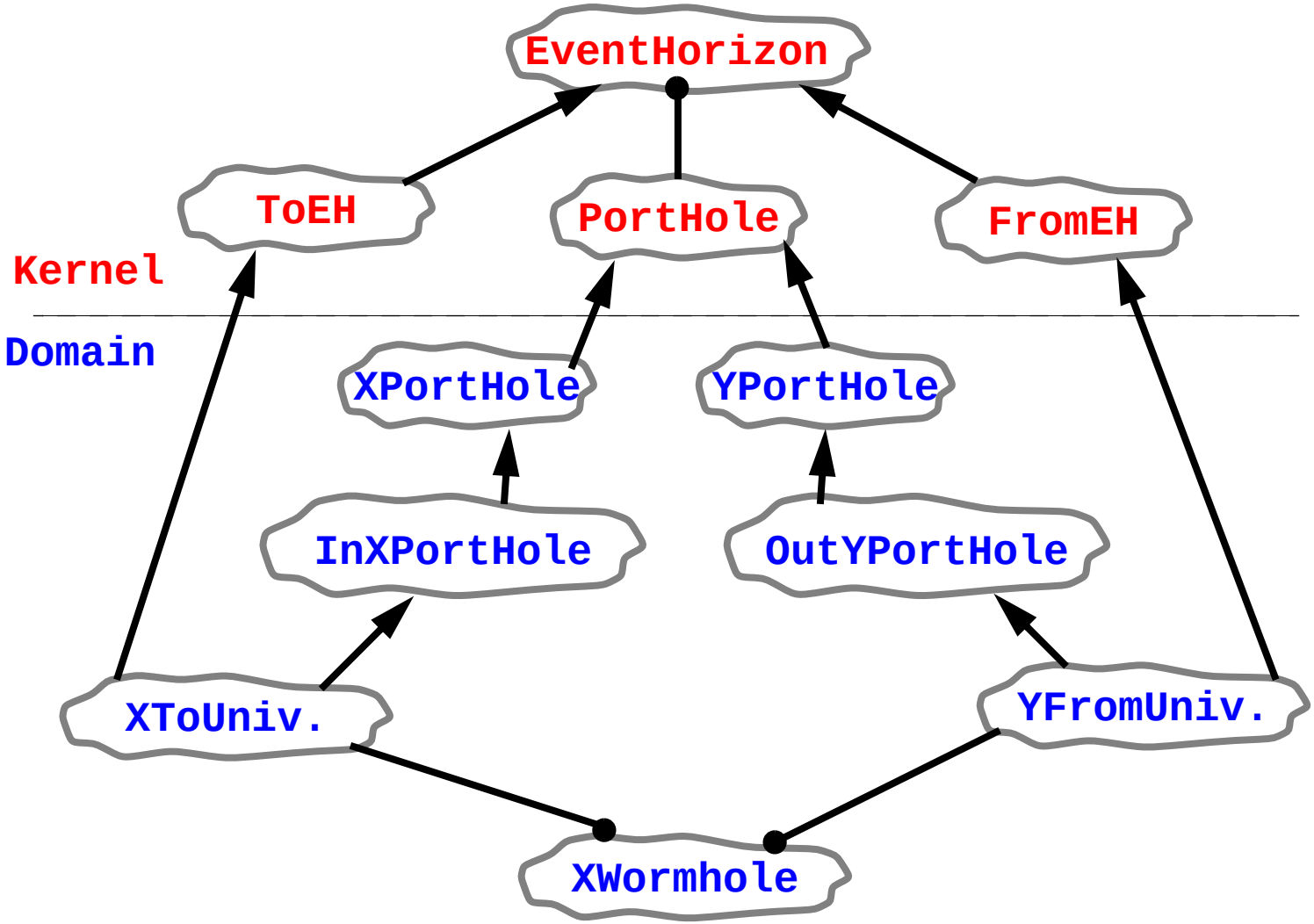
The following are probably not sufficient:

- It is too much trouble to understand the existing domains.
- You think the existing domains are wrong.
- You really want to write alot of C++ code.
- Writing domains is cool.

Domain-Specific Classes



More Domain-Specific Classes



Creating a domain ... an Overview

- **Scope out the computational specialization you need.**
  - what rules regulate the firing of actors?
  - what rules dictate how/when data is transferred between actors?
  - code generation domains - what supervisory functions are required?
  - are new data types needed?
- **Create outlines for new domain classes (see next slide).**
- **Map the functionality onto domain-specific classes derived from kernel classes.**
  - need to understand and accommodate mismatches
  - create any new classes required to support the new semantics
- **Consider semantic mapping issues between domains, too.**
  - how will/should the domain interact with others?
  - what semantics are built in vs. left for designer?

The Process... an Overview (cont.)

- **Create few basic Stars for the domain for testing.**
  - **debug basic semantics.**
- **Build up and debug a library of domain-specific Stars.**
- **Debug heterogeneous interactions.**

Mechanics: Mkdom and Domain Class Templates

- **To create domain-specific class templates:**

```
% cd ~ptolemy/src/domains/; mkdir new;
% cd new; mkdom new
```

- **Creates:**

```
make.template (!!)
    .../new/kernel
NEWDomain.cc (NEWDomain and default NEWTarget)
NEWGeodesic.h
NEWPortHole.h  NEWPortHole.cc
NEWScheduler.h  NEWScheduler.cc
NEWStar.h  NEWStar.cc
NEWWormhole.h  NEWWormhole.cc
    (NEWWormhole and NEWTo/NEWFromUniversal)

.../new/stars
NEWNop.pl
```

- **Compiles (!!), but does little more.**

Typical Domain Extensions

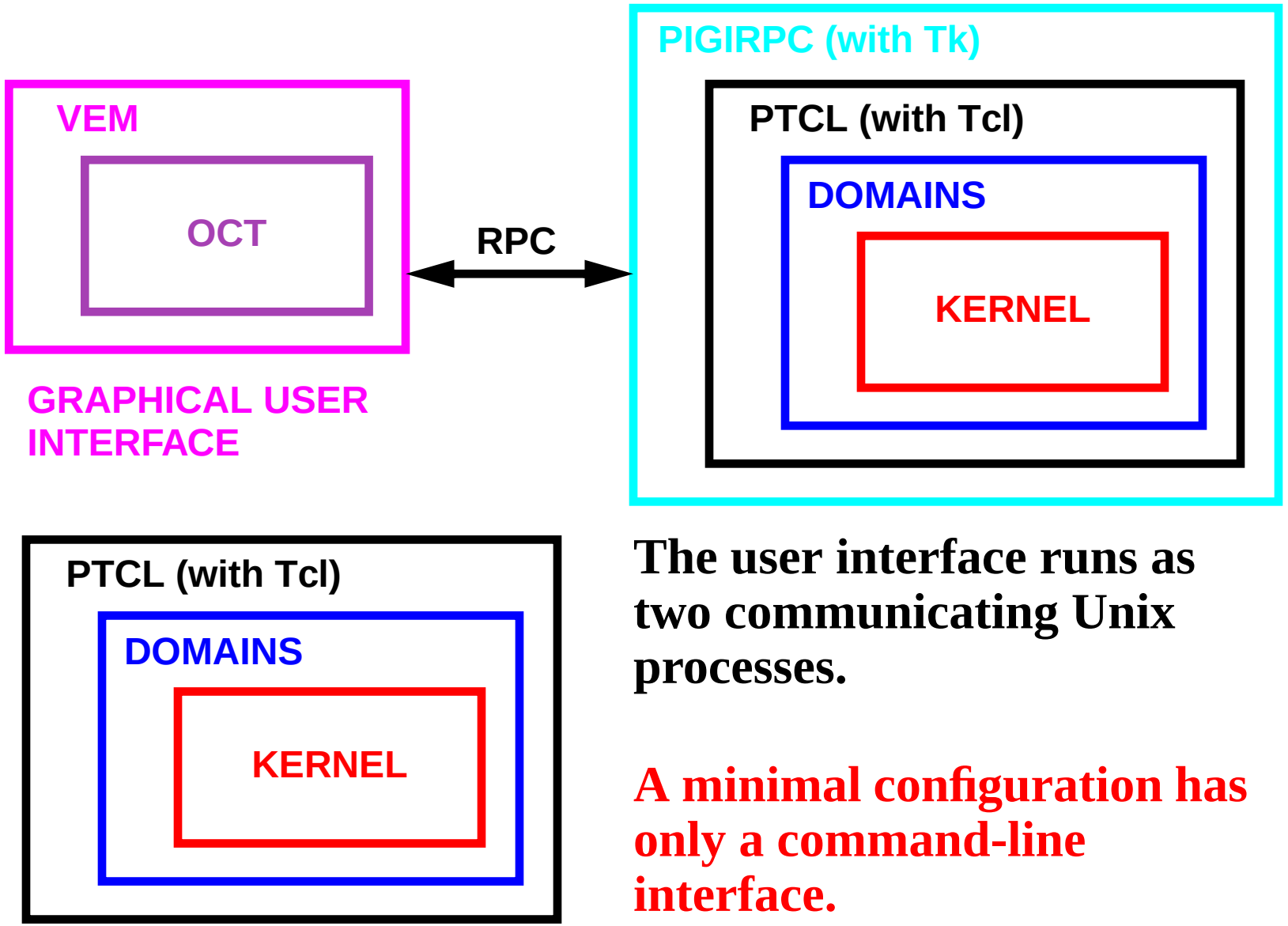
Since Ptolemy is OO, many extensions, specializations are possible; some are required to have an operational domain.

- **NEWScheduler - Sequences execution of blocks based on domain semantics.**
- **NEWPortHole - Supports data transfer mechanism for the domain.**
- **NEWStar - Enforces sequence of getting/sending data or events relative to execution. Specific functional stars inherit this behavior.**
- **NEWTo/NEWFromUniversal - Builds correct PortHole/ EventHorizon combos for NEWWormholes; implements Wormhole data transfer.**

Typical Domain Extensions (cont.)

- **NEWDomain - Supports creation of inter-domain mechanisms like NEWToUniversal and NEWFromUniversal, and NEWWormhole.**
- **NEWWormhole - Dual-natured building block at domain interfaces.**
- **NEWTarget - Supervises execution within target system; used by code generation domains mostly.**

Overall Software Organization



The user interface runs as two communicating Unix processes.

A minimal configuration has only a command-line interface.

Standard forms of the Ptolemy Executables

pigi

- All domains and targets plus the GUI.

ptrim

- SDF, BDF, DDF, DE, CGC and HOF domains plus GUI.

ptiny

- SDF and DE domains plus GUI.

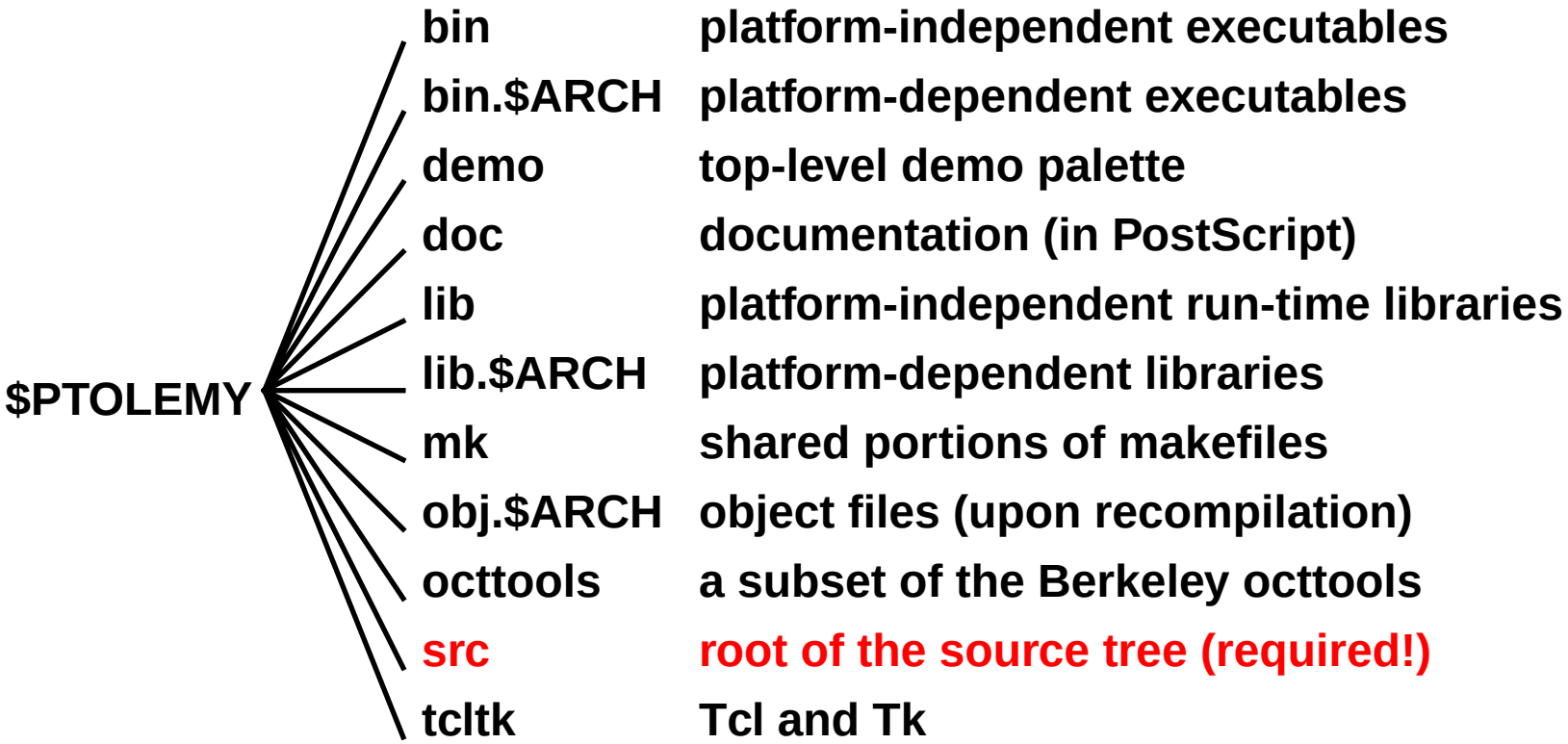
ptcl

- All domains, no GUI.

pitcl

- Experimental OO replacement for ptcl.

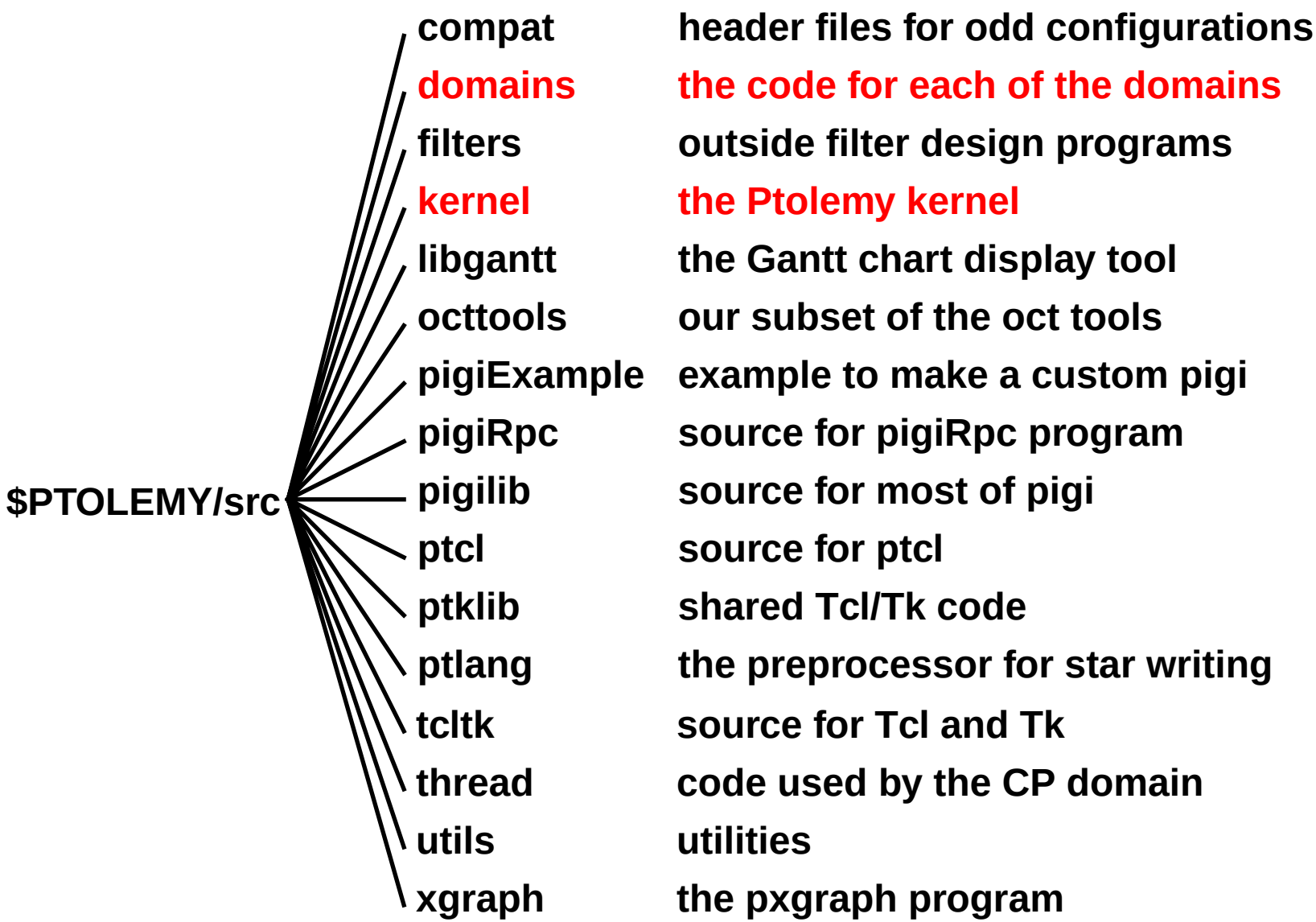
File Organization



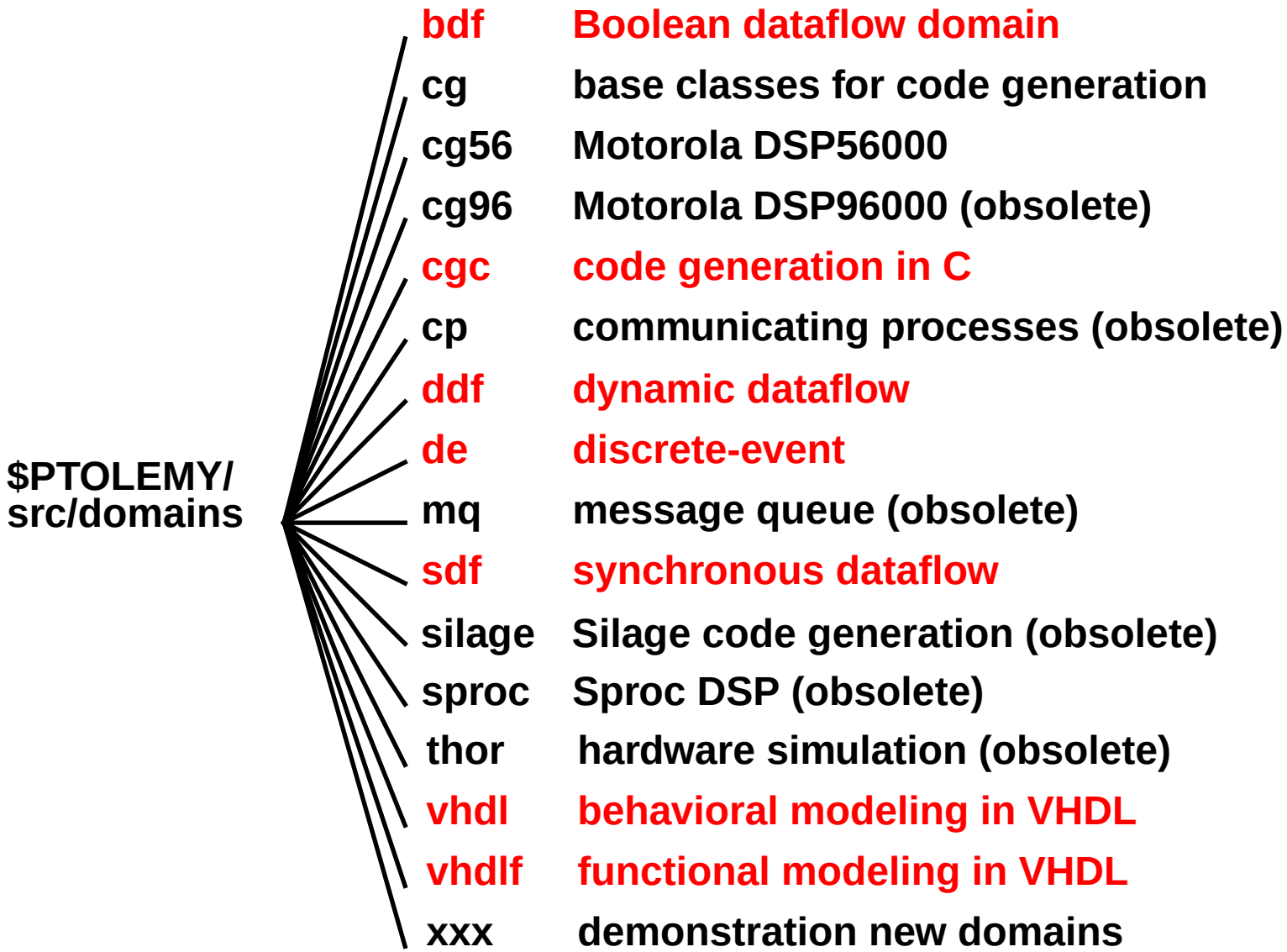
To run ptolemy, set the environment variable PTOLEMY to the root directory of the installation and put $PTOLEMY/bin in your path.

The preferred installation location is /users/ptolemy.

The Source Tree



The Domains Directory



Within Each Domain Directory

The stars in a domain are an excellent resource for learning about how to write custom stars for the domain.

READ THE SOURCE

Development Environment

Tools used:

- g++/gcc/gmake for compilation
- Cfront for portability validation
- Purify for memory management validation
- SCCS for version control
- Emacs + gdb for interactive debugging
- FrameMaker and HTML for documentation
- Quantify for performance measurements
- NFS for shared filesystem

Most researchers create parallel source and object trees that allow for unobtrusive experimentation with the code. These are connected to the SCCS database to manage multiple access and maintain consistency.

Debugging

- Dealing with core dumps
- Interpreting stack traces
- Running a symbolic debugger
- Creating custom executables

Note that if you are using Ptolemy as a software development environment (a reasonable use), you need to invest some effort in learning to use reasonable tools (make, gdb, emacs). Of these, only make is distributed with Ptolemy, but all are freely available on the net.

Core Dumps

Recall that Ptolemy runs as two processes:

- **vem**
- **pigiRpc**

Vem rarely crashes in normal usage, but since Ptolemy is extensible, pigiRpc is fairly likely to crash as you develop code (or discover bugs in our code). You may get any of several messages, among them:

```
RPC Error: server: application exited without calling
RPCExit
Closing Application /home/ohm1/users/messer/ptolemy/lib/
pigiRpcShell on host foucault.berkeley.edu
Elapsed time is 1538 seconds
```

You may get the messages:

```
segmentation fault (core dumped)
```

The vem interface can no longer issue Ptolemy commands.

Stack Traces

```
% gdb pigiRpc
    ...
Core was generated by 'pigiRpc'.
Program terminated with signal 11, Segmentation fault.
#0 0x2eebb4 in end ()
(gdb) where
#0 0x2eebb4 in end ()
#1 0x1c774c in SimControl::doPreActions ()
#2 0x1c8538 in Star::run ()
#3 0x16ee74 in DataFlowStar::run ()
#4 0x16d888 in SDFScheduler::runOnce ()
#5 0x16d7c4 in SDFScheduler::run ()
#6 0x1cb4c0 in Target::run ()
#7 0x1cd4e4 in Runnable::run ()
#8 0x1bd810 in InterpUniverse::run ()
#9 0x197774 in KcRun ()
#10 0x18edac in RpcRun ()
#11 0x18eeb8 in RpcReRun ()
#12 0x177634 in RPCApplicationProcessEvents ()
#13 0x17675c in RPCMain ()
#14 0x2300 in main ()
```

More Extensive Debugging

Make a debug version of the executable (e.g. on Solaris):

```
cd $PTOLEMY/obj.sol2/pigiRpc
make ptinyRpc.debug
```

Make the smallest debug version that contains the code you are testing (ptinyRpc, ptrimRpc, or pigiRpc).

Running the new executable:

```
setenv PIGIRPC $PTOLEMY/obj.sol2/pigiRpc/ptinyRpc.debug
pigi -debug
```

Optional, but strongly recommended (before the above, or in your .cshrc file):

```
setenv PT_DEBUG ptgdb
```

This specifies to use emacs and gdb (a powerful combination).

Creating your own Executable

Dynamic linking does not leave enough information to fully use symbolic debugging.

Create your own pigiRpc, ptrimRpc, or ptinyRpc that includes your stars.

An example is provided in $PTOLEMY/src/pigiExample. Copy the contents of this directory, replace your star with the example there (SDFMyStar.pl) and recompile:

```
make ptinyRpc.debug
```

You may optionally edit the file there "defpalettes.c" to change the default domain and the list of palettes you wish displayed. The make.template is extensively commented. READ IT.